Priority Queue based on multilevel prefix tree

David S. Płaneta*

February 1, 2008

Abstract

Tree structures are very often used data structures. Among ordered types of trees there are many variants whose basic operations such as insert, delete, search, delete-min are characterized by logarithmic time complexity. In the article I am going to present the structure whose time complexity for each of the above operations is $O(\frac{M}{K} + K)$, where M is the size of data type and K is constant properly matching the size of data type. Properly matched K will make the structure function as a very effective Priority Queue. The structure size linearly depends on the number and size of elements. PTrie is a clever combination of the idea of prefix tree – Trie, structure of logarithmic time complexity for insert and remove operations, doubly linked list and queues.

1 Introduction

Priority Trie (PTrie [5]) uses a few structures including Trie of 2^K degree [1], which is the structure core. Data recording in PTrie consists in breaking the word into parts which make the indexes of the following layers in the structure (table look-at). The last layers contain the addresses of doubly linked list's nodes. Each of the list nodes stores the queue [3], into which the elements are inserted. Moreover, each layer contains the structure of logarithmic time complexity of insert and remove operations. Which help to define the destination of data in the doubly linked list [3]. They can be various variants of ordered trees [3] or a skip list [4].

1.1 Terminology

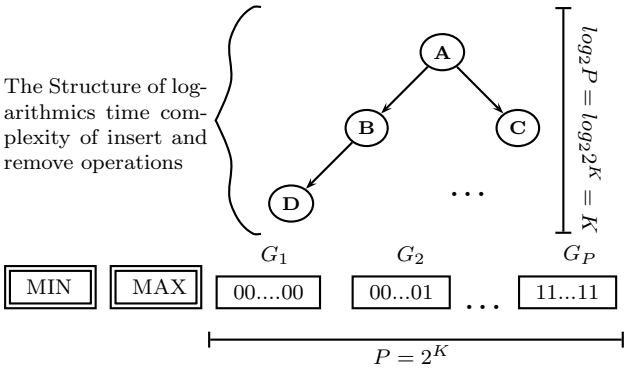
Bit pattern is a set of K bits. K (length of bit pattern) defines the number of bits which are cut off the binary word. M defines number (length) of bits in a binary word.

*dplaneta@gmail.com

$$\text{value of word} = \overbrace{\underbrace{101...1}_{K}00101...}^{M}$$

N is number of all values of PTrie. 2^K is variation K of element binary set $\{0, 1\}$. It determines the number of groups (number of Layers [Figure 1]), which the bit pattern may be divided into during one step (one level). The set of values decomposed into the group by the first K bits (the version of algorithm described in paper was implemented by machine of little-endian type). The

Figure 1: Layer



path is defined starting from the most important bits of variable. The value of pattern K (index) determines the layer we move to [Figure 2]. The lowest layers determine the nodes of the list which store the queues for inserted values. L defines the level the layer is on. Probability that exactly G keys correspond to one particular pattern, where for each of P_L sequences of leading bits there is such a node that corresponds to at least two keys equals

$$\binom{N}{G} P^{-GL}(1 - P^{-L})^{N-G}$$

For random PTrie the average number of layers on level L, for $L = 0, 1, 2, \ldots$is

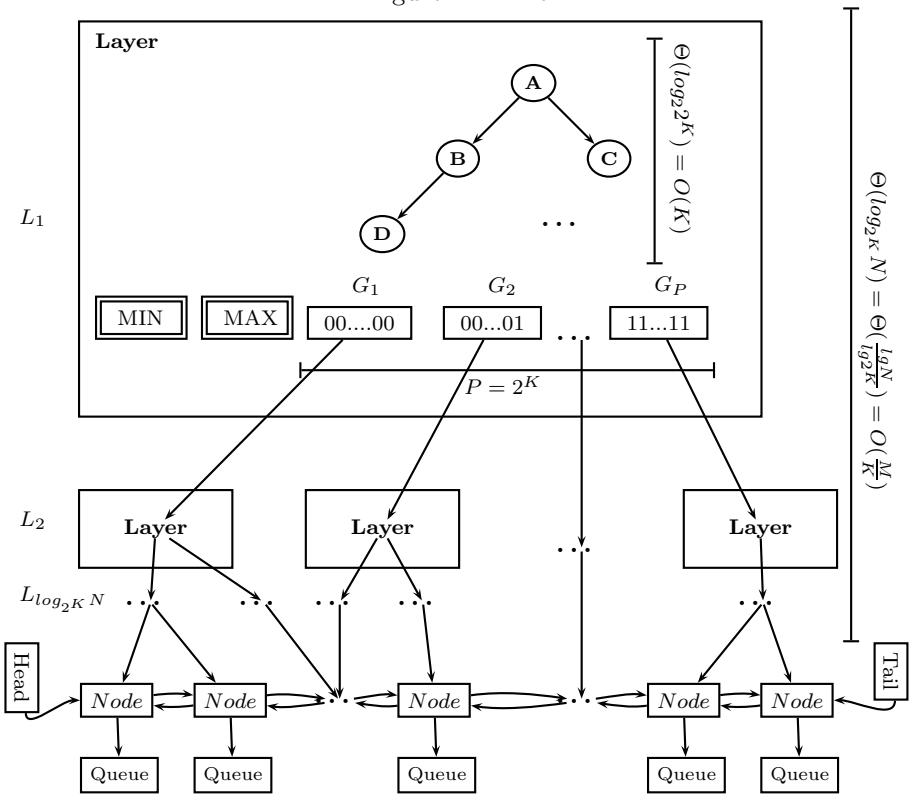
$$P^L(1 - (1 - P^{-L})^N) - N(1 - P^{-L})^{N-1}$$

If A_N is average number of layers in random PTrie of degree $P = 2^K$ containing N keys. Then $A_0 = A_1 = 0$, and for $N \geq 2$ we get [2]:

$$A_N = 1 + \sum_{G_1+\ldots+G_P=N} \Big(\frac{N!}{G_1! \ldots G_P!} P^{-N}\Big)\Big(A_{G_1} + \ldots + A_{G_P}\Big) =$$

$$1 + P^{1-N} \sum_{G_1+\ldots+G_P=N} \Big(\frac{N!}{G_1! \ldots G_P!}\Big) A_{G_1} =$$

Figure 2: PTrie



$$1 + P^{1-N} \sum_{G} \binom{N}{G} \Big(P-1\Big)^{N-G} A_G =$$

$$1 + 2^{G(1-N)} \sum_{G} \binom{N}{G} \Big(2^G-1\Big)^{N-G} A_G$$

2 Implementation

Operation	**Description**	**Bound**
create	Creates object	$O(1)$
insert(data)	Adds element to the structure.	$O(\frac{M}{K} + K)$
boolean remove(data)	Removes value from the tree. If operation failed because there was no such value in the tree it returns FALSE(0), otherwise returns TRUE(0).	$O(\frac{M}{K} + K)$
boolean search(data)	Looks for the words in the tree. If finds return TRUE(1), otherwise FALSE(0).	$O(\frac{M}{K})$
*minimum()	Returns the address of the lowest value in the tree, or empty address if the operation failed because the tree was empty.	$O(1)$
*maximum()	Returns the address of the highest value in the tree or empty address if the operation failed because the tree was empty.	$O(1)$
next	Returns the address of the next node in the tree or empty address if value transmitted in parameter was the greatest. The order of moving to successive elements is fixed - from the smallest to the largest and from "the youngest to the oldest" (stable) in case of identical words.	$O(1)$
back	Similar to 'next' but it returns the address of preceding node in the tree.	$O(1)$

Basic operations can be joined. For example, the effect connected with the heap; delete-min() can be replaced by operations remove(minimum()).

2.1 Insert

Determine the interlinked index (pointer) to another layer using the length of pattern projecting on the word.

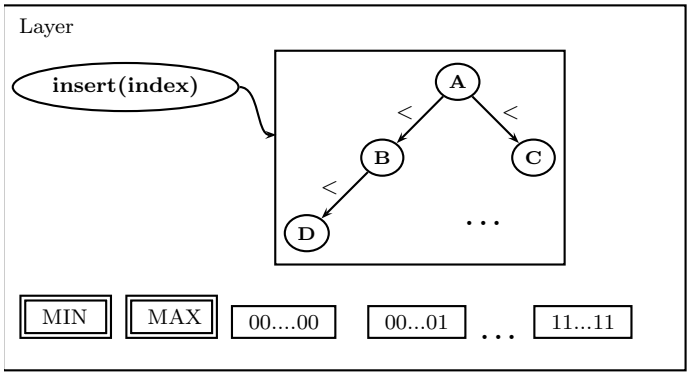
If interlink determined by index is not empty and indicated the list node –

try to insert the value into the queue of determined node.
If the elements in the queue turn out to be the same, insert value into the queue. *Otherwise*, if elements in the queue are different from the inserted value, the node is "pushed" to a lower level and the hitherto existing level (the place of node) is complemented with a new layer. Next, try again to insert the element, this time however, into the newly created layer.

Else, if the interlink determined by index is empty, insert value of index into the ordered binary tree from the current layer [Figure 3]. Father of a newly created node in ordered binary tree from the current layer determines the place for leaves; If the newly created node in ordered binary tree is on the right side of father (added index > father index), the value added to the list will be inserted after the node determined by father index and the path of the highest indexes (make use of pointer 'max' of the layers – time cost $O(1)$) of lower level layers. If newly created node is on the left side of father (added index < father index), the value added to the list will be inserted before the node determined by father index and the path of the smallest indexes (make use of pointer 'min' of the layers – time cost $O(1)$) of lower level layers.

One can wonder why we use the queue and not the stack or the value

Figure 3: Insert value of index into the ordered binary tree from the layer



counter. Value counter cannot be used because complex elements can be inserted into PTrie structure, distinguishable in the tree only because of some words. Also, it is not a good idea to use a stack because the queue makes the structure stable. And this is a very useful characteristic. I used "plain" Binary Search Tree in the structure of logarithmic time complexity. For a small number of tree nodes it is a very good solution because for $K = 4$, $2^K = 16$. So in the tree there may be maximum 16 (different) elements. For such a small amount of (different) values the remaining ordered trees will probably turn out to be at most as effective as unusually simple Binary Search Trees.

2.1.1 Analysis

In case of random data it will take $\Theta(\frac{lgN}{lg2^K}) = \Theta(log_{2^K}N) = O(\frac{M}{K})$ goings through layers to find the place in the heap core – Trie tree. On at least one layer of PTrie structure we will use inserting into the ordered binary tree in which maximum number of nodes is 2^K. While inserting the new value I need information where exactly it will be located in the list. Such information can be obtained in two ways; I will get the information if the representation of the nearest index on the list is to the left or to the right side of the inserted word index. It may happen that in the structure there is already is exactly the same word as the inserted one. In such case value index won't be inserted into any layer of the PTrie because it will not be necessary to add a new node of the list. Value will be inserted into the queue of already existing node. To sum up, while moving through the layers of PTrie we can stop at some level because of empty index. Then, a node will be added to the list in place determined by binary search tree and the remaining part of the path. This is why the bound of operation which inserts new value into PTrie equals $\Theta(log_{2^K}N+log_2 2^K) = \Theta(log_{2^K}N+K) = O(\frac{M}{K}+K)$.

2.2 Find

Method find like in case of plain Trie trees goes through succeeding layers following the path determined by binary representation of search value. It can be stated that it uses number key as a guide while moving down the core of PTrie – prefix tree. In case of searching tree things can happen:

- We don't reach the node of the list because the index we determine is empty on any of layers – searching failure.
- We reach the node but values from the queue are different from the searched value – searching failure.
- We reach the node and the values from the queue are exactly like the ones we seek – searching success.

2.2.1 Analysis

Searching in prefix tree is very fast because it finds the words using word key as indexes. In case of search failure the longest match of a searched word is found. It must be taken into consideration that during operation 'search' we use only the attributes of prefix tree. This is why the amount of search numbers looked through during the random search is $\Theta(log_{2^K}N) = O(\frac{M}{K})$.

2.3 Remove

Remove method just like find method "moves down" the PTrie structure to seek for the element to be deleted. If it doesn't reach the node of the list, or it does but the search value is different from the value of node queue, it does not delete any element of PTrie because it is not there. However if it reaches the node of the list and search value turns out to be the value from the queue – it removes the value from the queue. If it remains empty after removing the element from the queue the node will be removed from the list and will return to the "upper" layers of prefix tree to delete possible, remaining, empty layers.

2.3.1 Analysis

Since it is possible not only to go down the tree but also come back upwards (in case of deleting of the lower layer or the node of the list) the total length of the path move on is limited $\Theta(2log_{2^K}N)$. If delete the layer, it means there was only one way down from that layer, which implicates the fact that the ordered binary tree of a given layer contained only one node (index). The layer is removed if it remains empty after the removal of node from ordered binary tree. So the number of operation necessary for the removal of the layer containing one element equals $\Theta(1)$. In case of removal of layer L_i, if ordered binary tree of higher level layer L_{i-1}, despite removing the node which determines empty layer we came from, does not remain empty it means that there could be maximum 2^K nodes in the ordered binary tree. Operation of value delete from ordered binary tree amounts to $\Theta(log_2 2^K) = \Theta(K)$. There is no point of "climbing" up the upper layers, since the layer we came from would not be empty. At this stage the method remove ends. To sum up, worse time complexity of remove operation is $\Theta(2log_{2^K}N + K) = O(\frac{M}{K} + K)$.

2.4 Extract minimum and maximum

If the list is not empty, 'minimum' reads the value pointed by the head of the list and 'Maximum' reads the value pointed by the tail of the list.

2.4.1 Analysis

Time complexity of operations is $\Theta(1)$.

2.5 Iterators

The nodes of the list are linked. If we know the position of one of the nodes, we have a direct access to its neighbors. The 'next' operation reads the successor of current pointed node. The 'prev' operation reads the predecessor of currently pointed node.

2.5.1 Analysis

Moving to the node its neighbor requires only reading of the contents of the pointer 'next' or 'prev'. Time complexity of such operations equals $\Theta(1)$.

3 Correctness

PTrie has been designed like this, so as not to assume that keys have to be positive numbers or only integers - they can be even strings (however, in most cases the weight of arcs is represented by numbers). To insert PTrie negative and positive integers I use not one PTrie, but two! One of the structures is destined exclusively for storing positive integers and the other one for storing only negative integers. The latter structure of PTrie is responsible only for negative integers - the integers are stored in reverse order on the list (for machine of little-endian type). Therefore in case of the second structure of PTrie (responsible only for negative integers) I used standard operation of PTrie: PTrie2.maximum to extract the smallest value. Also real numbers (for example in ANSI IEEE 754-1985 standard) can be used of the description of the weight of arcs on condition that two interrelated structures of PTrie will be used to put off exponent and mantissa. It is possible, because implementation of PTrie described by me uses queue, which makes it stable. One of the structures of PTrie serves as storage for exponent, where each of the nodes of the list will contain additional structure of PTrie to store mantissa.

4 Conclusions

Efficiency of PTrie (source codes [5]) considerably depends on the length of pattern K. K defines optional value, which is the power of two in the range $[1,\ \min(M)]$. The total size of necessary memory bound is proportional to $\Theta(\frac{log_{2^K} N(2^{K+1})}{K})$ because the number of layers required to remember N random elements in PTrie of degree 2^K equals $\frac{lgN}{lgP} * P$. Moreover, each layers has tree of maximum size 2^K nodes and table of the P-elements, so the necessary memory bound equal $\Theta(log_{2^K} N * 2P) = \Theta(\frac{M}{K} * 2^{K+1})$. For data types of constant size maximum Trie tree height equals $\frac{M}{K}$. So the pessimistic operation time complexity is $O(\frac{M}{K} + K)$. For example, for four-byte numbers it is the most effective to determine the pattern $K = 4$ bits long. Then, the pessimistic number of steps necessary for the operation on the PTrie will equal $\Theta(\frac{M}{K} + K) = \frac{32}{4} + 4 = 12$. Increasing K to $K = 8$ does not increase the efficiency of the structure operation because $\Theta(\frac{M}{K} + K) = \frac{32}{8} + 8 = 12$. What is more, in will unnecessarily increase the memory demand. A single layer consisting of $P = 2^K$ groups for $K = 8$ will contain tables $P = 2^8 = 256$ long, not when $K = 4$, only $P = 2^4 = 16$ links. For variable size data the time complexity equals $\Theta(log_{2^K} N + K)$. Moreover, the length of pattern K must be carefully matched. For example, for strings K should not be longer than 8 bits because we could

accidentally read the contents from beyond the string which normally consist of one-byte sign! It is possible to record data of variable size in the structure provided each of the analyzed words will end with identical key. There are no obstacles for strings because they normally finish with "end of line" sign. Owing to the reading of word keys and going through indexes (table look-at), primary, partial operations of PTrie method are very fast. If we carefully match K with data type, PTrie will certainly serve as a really effective Priority Queue [6].

References

[1] René de la Briandais, *File Searching Using Variable Length Keys*, Proceedings of the Western Joint Computer Conference, 295-298, 1959.

[2] Donald E. Knuth, *The Art of Computer Programming* Vol. 3, Addison Wesley Longman, Inc. 1998.

[3] Donald E. Knuth, *The Art of Computer Programming* Vol. 1, Addison Wesley Longman, Inc. 1998.

[4] William Pugh, *Skip lists are a data structure that can be used in place of balanced trees*, Communications of the ACM, 33(6) 668-676, June 1990.

[5] David S. Płaneta, *PTrie: Priority Queue based on multilevel prefix tree*, Cornell University Computing and Information Science Technical Reports, TR2006-2023, 2006. [Online]. Available: `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2006-2023` Available [source]: `http://http://ptrie.sourceforge.net`

[6] David S. Płaneta, *Linear Time Algorithms Based on Multilevel Prefix Tree for Finding Shortest Path with Positive Weights and Minimum Spanning Tree in a Networks*, Cornell University Computing and Information Science Technical Reports, TR2006-2043, 2006. [Online]. Available: `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2006-2043`